Московский государственный университет имени М.В. Ломоносова
Факультет вычислительной математики и кибернетики
Кафедра суперкомпьютеров и квантовой информатики

Мазеев Артём Валерьевич

Разработка и исследование параллельного алгоритма поиска минимального остовного дерева в графе

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА

Научный руководитель
к.ф.-м.н., доцент
Попова Н.Н.

Научный консультант
к.т.н.
Семенов А.С.

Москва 2016

Содержание

Аннотация

В данной работе рассматривается проблема поиска минимального остовного дерева в больших графах. Задача поиска минимального остовного дерева встречается во многих областях, например, в компьютерном зрении, биоинформатике, а также при проектировании различных сетей, причем часто возникает необходимость обработки больших графов. Под большими подразумеваются графы, которые не могут быть представлены в оперативной памяти типичного вычислительного узла многопроцессорной вычислительной системы. На сегодняшний день объем памяти узла в среднем составляет 64–128 ГБ, такому объему памяти соответствует граф порядка 100 миллионов вершин и нескольких миллиардов ребер.

Целью работы является разработка и исследование параллельного алгоритма поиска минимального остовного дерева для вычислительных систем с распределенной памятью. Для параллельной реализации на распределенной памяти существует несколько алгоритмов, среди которых базовым является алгоритм GHS, поэтому он был взят за основу. Алгоритм основан на передаче сообщений между смежными вершинами графа.

В работе разработан параллельный алгоритм построения минимального остовного дерева. Автором подробно изложены проблемы, возникающие при разработке такого алгоритма, а также методы решения этих проблем. Часть работы посвящена исследованию алгоритма. Для оценки производительности программной реализации используются RMAT графы.

Разработка и исследование алгоритма производились на 36-узловом вычислительном кластере «Ангара-К1», с использованием языка программирования C++ и библиотеки передачи сообщений MPI. Кроме того, на

1024 узлах суперкомпьютера IBM Blue Gene/P продемонстрирована сильная масштабируемость разработанной реализации. Основные результаты работы описаны в заключении.

1 Введение

1.1 Актуальность

Задача поиска минимального остовного дерева встречается при решении многих задач, например, при проектировании различных сетей (телефонных, электрических, компьютерных и т. д.), с целью минимизации общей стоимости проекта. Кроме того, задача поиска минимального остовного дерева возникает также при анализе финансовых рынков, в компьютерном зрении и биоинформатике.

В реальных задачах постоянно растут требования к объему обрабатываемых графов. Например, в биоинформатике для решения задачи кластеризации [1], которая может быть решена при помощи построения минимального остовного дерева, необходимые для обработки графы могут занимать память петабайтного и более объема. В связи с этим проблема построения минимального остовного дерева для больших графов является актуальной. Под большим в данной работе подразумевается граф, который не может быть представлен в оперативной памяти типичного вычислительного узла многопроцессорной вычислительной системы. На сегодняшний день объем памяти узла в среднем составляет 64–128 ГБ, такому объему памяти соответствует граф порядка 100 миллионов вершин и нескольких миллиардов ребер без учета дополнительной памяти, необходимой для решения задачи.

1.2 Описание проблемы

Пусть $G = (V, E)$ — взвешенный неориентированный граф, V — множество вершин, E — множество ребер. $|V| = N, |E| = M$. Остовное

дерево (в связном графе) — ациклический связный подграф данного связного неориентированного графа, в который входят все его вершины. Иными словами, остовное дерево в связном неориентированном графе — дерево в этом графе, которое содержит все вершины. Минимальное остовное дерево MST (англ. Minimum Spanning Tree) [2] в связном взвешенном неориентированном графе — это остовное дерево этого графа, имеющее минимальный возможный вес, где под весом дерева понимается сумма весов входящих в него ребер. Остовный лес (в несвязном графе) — подграф, состоящий из объединения остовных деревьев для каждой его компоненты связности. Соответственно, минимальный остовный лес — подграф, состоящий из объединения минимальных остовных деревьев для каждой его компоненты связности.

Иногда бывает необходимо найти минимальное остовное дерево в большом графе. Для решения этой проблемы нужен параллельный алгоритм поиска минимального остовного дерева для вычислительной системы с распределенной памятью.

В данной работе требуется разработать и исследовать параллельный алгоритм поиска минимального остовного дерева в графе для вычислительных систем с распределенной памятью.

2 Обзор алгоритмов поиска минимального остовного дерева в графе

Существует большое количество алгоритмов [3], решающих задачу MST. Некоторые из алгоритмов приведены в таблице 1. Алгоритмы из верхней половины таблицы, по своей сути, являются последовательными. Среди них наилучшей асимптотикой обладает алгоритм Chazelle.

Алгоритм Boruvka [4] удобен для распараллеливания на общей памяти, однако для параллельной реализации на распределенной памяти разработаны специальные алгоритмы, например, алгоритм GHS (англ. Gallager, Humblet, Spira) [5], Awerbuch [6] и другие. Алгоритм Awerbuch является развитием и оптимизацией алгоритма GHS. Поэтому GHS был выбран для исследования как базовый и более простой.

Таблица 1. Обзор алгоритмов.

Алгоритм	**Год создания**	**Сложность**
Prim [7]	1957	$O(M + N * logN)$
Kruskal [8]	1956	$O(M * logN)$
Reverse-delete [8]	1956	$O(M * logN * (loglogN)^3)$
Chazelle [9]	2000	$O(M * \alpha(M, N))$, где α — обратная функция Аккермана
Boruvka [4]	1926	$O(M * logN)$
Bader and Cong [10]	2006	$O(M * logN/p)$, где p — количество процессоров
GHS [5]	1983	$O(N * logN), p = N$
Awerbuch [6]	1987	$O(N), p = N$

Есть множество реализаций различных алгоритмов для общей памяти, например [11–14], а также для распределенной памяти — [15–19]. Только в одной статье [18] есть исследование алгоритма GHS, однако информации

в этой статье недостаточно для обоснованного сравнения результатов.

2.1 Описание алгоритма GHS

Алгоритм GHS очень хорошо описывается в модели vertex-centric [20]. Идея алгоритма состоит в следующем. Все вершины графа выполняют одну и ту же процедуру, которая заключается в отправке, приеме и обработке сообщений от смежных вершин. По каждому ребру графа сообщения передаются независимо в обоих направлениях, причем они не должны обгонять друг друга в рамках одного направления по ребру.

В каждый момент времени множество вершин графа представляется в виде объединения некоторого количества фрагментов, то есть непересекающихся множеств вершин. Изначально каждая вершина является фрагментом. Каждый фрагмент находит среди ребер, исходящих из него в другие фрагменты, ребро с минимальным весом (минимальное исходящее ребро). Далее, по этим ребрам фрагменты объединяются. Ребра, по которым происходят объединения фрагментов, будут составлять минимальное остовное дерево, когда для связного графа останется ровно один фрагмент, включающий в себя все вершины.

Рассмотрим работу алгоритма подробнее. Существует три возможных состояния вершины: *Sleeping* — начальное состояние, *Find* — когда вершина участвует в поиске минимального исходящего из фрагмента ребра, и *Found* — в остальных случаях. У каждого фрагмента есть переменная L, характеризующая его уровень. Изначально уровень каждого фрагмента равен 0. Два фрагмента с одинаковым уровнем L могут объединиться во фрагмент с уровнем $L+1$. Фрагмент не может присоединиться к фрагменту с меньшим уровнем.

Опишем подробнее, как происходит поиск минимального исходящего

ребра во фрагменте. В тривиальном случае, когда фрагмент состоит из одной вершины, и его уровень равен 0, вершина локально находит минимальное исходящее ребро, помечает его как часть минимального остовного дерева и отправляет сообщение *Connect* по этому ребру и переходит в состояние *Found*.

Теперь рассмотрим случай, когда уровень фрагмента больше 0. Предположим, что два фрагмента уровня $L-1$ объединились во фрагмент уровня L по одному исходящему ребру. Ребро, по которому объединились два фрагмента, становится ядром нового фрагмента. Вес ядра используется в качестве идентификатора фрагмента. Далее сообщение *Initiate* рассылается по всему фрагменту, начиная от вершин, смежных с ядром. Сообщение *Initiate* рассылается по фрагменту для того, чтобы все вершины получили новый уровень и идентификатор фрагмента, а также установили свое состояние в *Find*. Когда вершина получает сообщение *Initiate*, она начинает участвовать в поиске минимального исходящего ребра.

Каждое ребро графа может быть в одном из трех состояний: *Branch* — ребро принадлежит минимальному остовному дереву, *Rejected* — не принадлежит, *Basic* — в случае, если пока не известно, будет это ребро принадлежать минимальному остовному дереву или нет. Чтобы найти минимальное исходящее ребро, в некоторой вершине v поочередно перебираются, начиная с самого легкого, ребра, находящиеся в состоянии *Basic*. Для каждого такого ребра происходит проверка при помощи посылки по нему сообщений типа *Test*. Сообщение *Test* передает уровень и идентификатор фрагмента. Когда вершина u принимает сообщение *Test*, она сравнивает свой идентификатор фрагмента с принятым в сообщении. Если идентификаторы равны, тогда вершина u отправляет в ответ сообщение *Reject*, и после этого обе вершины устанавливают состояние ребра в *Reject*. В этом случае вершина v, которая отправила сообщение

Test, продолжает поиск, анализируя следующее наилучшее ребро и так далее. Если идентификатор фрагмента в полученном сообщении *Test* отличен от идентификатора фрагмента принимающей вершины u, и если уровень фрагмента у принимающей вершины больше либо равен, чем в сообщении *Test*, тогда в ответ отправляется сообщение *Accept*. Состояние ребра у вершины v в этом случае меняется на *Branch*. Однако если уровень фрагмента вершины u меньше, чем в пришедшем сообщении, тогда сообщение откладывается, пока уровень фрагмента принимающей вершины u не поднимется до нужного значения.

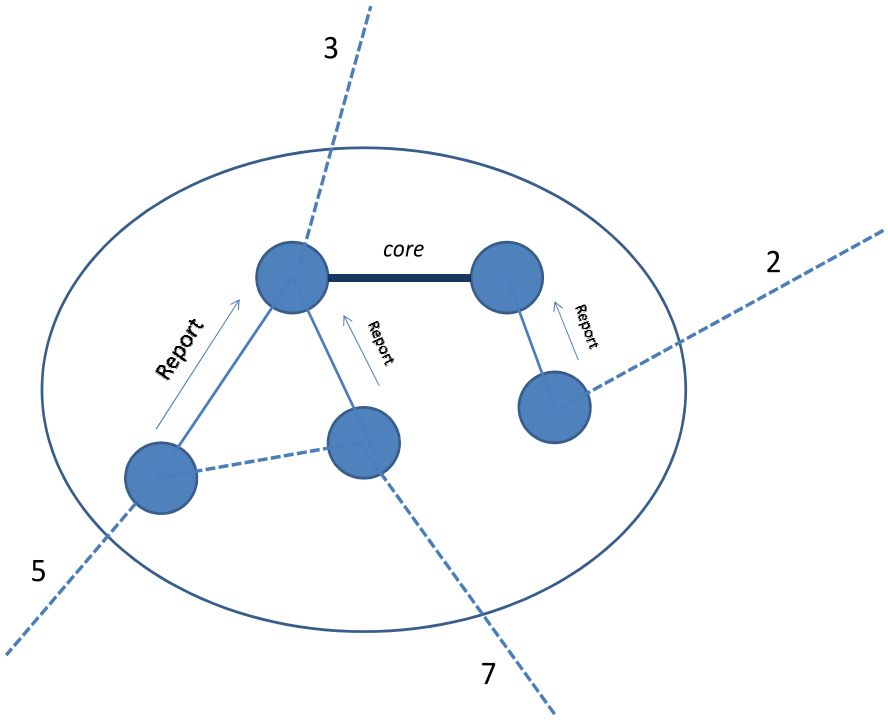

Рис. 1. Схема работы алгоритма GHS, отправка сообщений *Report*. Ребра, которые представлены сплошными линиями, находятся в состоянии *Branch*.

В конечном итоге, каждая вершина находит минимальное исходящее ребро, если такое имеется. Теперь вершины посылают сообщения *Report* (см. рис. 1), чтобы найти минимальное исходящее ребро всего фрагмента. Если ни одна из вершин графа не имеет исходящих ребер в состоянии

Basic, то алгоритм завершается, а ребра в состоянии *Branch* являются минимальным остовным деревом.

Отправка *Report* происходит по следующим правилам. Каждая листовая вершина фрагмента отправляет сообщение *Report(w)* по единственному инцидентному ребру в состоянии *Branch* (w — вес минимального исходящего ребра из вершины или бесконечность, если исходящих ребер нет). Каждая внутренняя вершина находит свое собственное минимальное исходящее ребро и ждет прибытия всех сообщений от всех поддеревьев. Затем вершина выбирает минимальный вес из всех значений весов. Если минимум достигается на значении, которое пришло из поддеревьев, то в переменную вершины *best_edge* записывается номер исходящей ветви (поддерева), иначе записывается номер минимального исходящего ребра. Это делается для того, чтобы в дальнейшем можно было легко восстановить путь, двигаясь туда, куда указывает *best_edge*. Далее происходит отправка *Report* вверх по дереву фрагмента с аргументом, равным уже найденному минимальному значению среди всех значений весов. Когда вершина отправляет сообщение *Report*, она также переходит в состояние *Found*. В конечном итоге, две вершины, инцидентные ядру, отправляют сообщения *Report* вдоль ядра и определяют вес минимального исходящего ребра, и с какой стороны от ядра оно находится.

Чтобы попытаться присоединить фрагмент к другому по найденному минимальному исходящему ребру фрагмента, можно воспользоваться переменной *best_edge* в каждой вершине, чтобы проследить путь от ядра до минимального исходящего ребра. Для этого от одной из ядровых вершин, которая ближе к минимальному исходящему ребру, посылается сообщение *Change core* (см. рис. 2). Вершина, получившая это сообщение, посылает его дальше в соответствии со своим значением *best_edge*, и так далее. Когда сообщение достигает вершины с минимальным исходящим ребром,

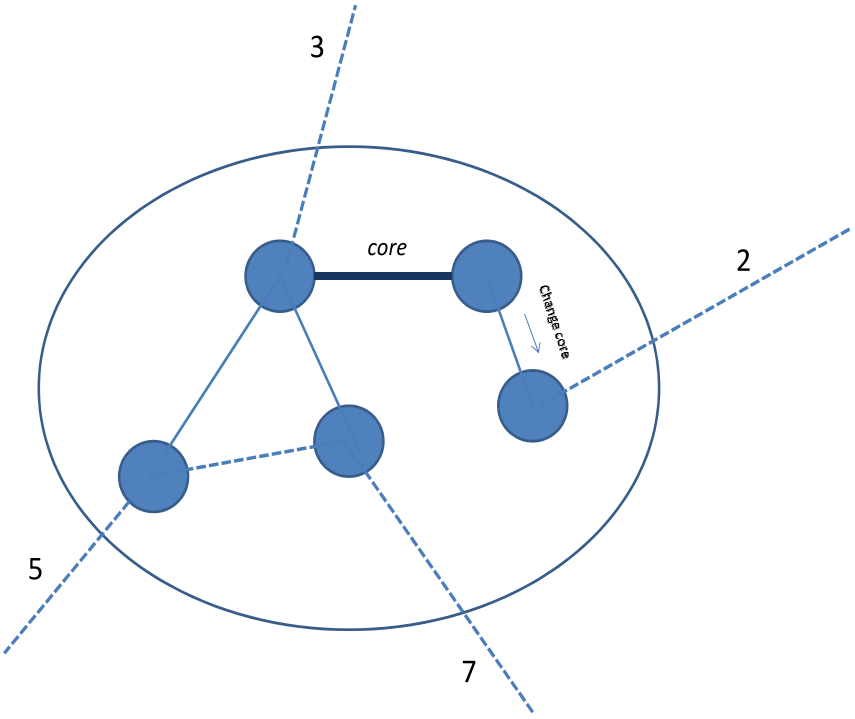

Рис. 2. Схема работы алгоритма GHS, отправка сообщения *Change core* в сторону с минимальным исходящим ребром фрагмента.

корнем дерева, которое образует фрагмент, становится эта вершина. Эта вершина посылает сообщение *Connect(L)* по минимальному исходящему ребру, где L — это уровень фрагмента. Если два фрагмента уровня L имеют одно и то же минимальное исходящее ребро, то каждый из них отправляет сообщение *Connect(L)* вдоль этого ребра, и это ребро становится ядром нового фрагмента уровня $L + 1$, которое сразу начнет рассылать по всему фрагменту сообщение *Initiate*, с новым номером уровня и идентификатором.

Когда сообщение *Connect* отправляет фрагмент с уровнем L и идентификатором F, во фрагмент с уровнем $L' > L$ и идентификатором F'. Больший фрагмент отправит сообщение *Initiate* с L' и F' в меньший фрагмент. В данном описании разобраны не все возникающие случаи, а лишь основные.

Сложность алгоритма (количество коммуникационных сообщений) — $O(N * logN)$.

В следующем подразделе приведено формальное описание алгоритма, которое состоит из процедур, выполняющихся при возникновении различных событий.

2.1.1 Псевдокод алгоритма GHS

Ниже в тексте приведены правила алгоритма GHS, которые выполняются в каждой вершине.

(1) Реакция на спонтанное пробуждение (только если вершина находится в состоянии *Sleeping*)

wakeup ();

(2) **procedure** wakeup

m = инцидентное ребро минимального веса;
$SE[m] = Branch$;
$LN = 0$;
$SN = Found$;
$Find\_count = 0$;
отправить сообщение $Connect(O)$ по ребру m;

(3) Реакция на сообщение $Connect(L)$ полученное по ребру j

if $SN = Sleeping$ **then**
  wakeup ();
if $L < LN$ **then**
  $SE[j] = Branch$;
  отправить сообщение $Initiate(LN, FN, SN)$ по ребру j;
  if $SN = Find$ **then**
    $Find\_count = Find\_count + 1$;
else if $SE[j] = Basic$ **then**
  поместить полученное сообщение в конец очереди;
else
  отправить сообщение $Initiate(LN + 1, w[j], Find)$ по ребру j;

(4) Реакция на сообщение $Initiate(L, F, S)$ полученное по ребру j

$LN = L; FN = F; SN = S; in\_branch = j;$
$best\_edge = nil; best\_wt = \infty;$
for *all* $i \neq j$ *таких что* $SE[i] = Branch$ **do**
  отправить сообщение $Initiate(L, F, S)$ по ребру i;
  if $S = Find$ **then**
    $Find\_count = Find\_count + 1;$
if $S = Find$ **then**
  test ();

(5) **procedure** test

if *существует хотя бы одно инцидентное ребро в состоянии* $Basic$ **then**
  $test\_edge =$ инцидентное ребро минимального веса в состоянии $Basic$;
  отправить сообщение $Test(LN, FN)$ по ребру $test\_edge$;
else
  $test\_edge = nil$;
  report ();

(6) Реакция на сообщение $Test(L, F)$ полученное по ребру j

if $SN = Sleeping$ **then**
  wakeup ();
if $L > LN$ **then**
  поместить полученное сообщение в конец очереди;
else if $F \neq FN$ **then**
  отправить сообщение $Accept$ по ребру j;
else
  if $SE[j] = Basic$ **then**
    $SE[j] = Rejected$;
  if $test\_edge \neq j$ **then**
    отправить сообщение $Reject$ по ребру j;
  else
    test ();

(7) Реакция на сообщение *Accept* полученное по ребру j

$test\_edge = nil$;
if $w[j] < best\_wt$ **then**
  $best\_edge = j$;
  $best\_wt = w[j]$;
report ();

(8) Реакция на сообщение *Reject* полученное по ребру j

if $SE[j] = Basic$ **then**
  $SE[j] = Rejected$;
test ();

(9) **procedure** report

if $Find\_count = 0$ *and* $test\_edge = nil$ **then**
  $SN = Found$;
  отправить сообщение $Report(best\_wt)$ по ребру in_branch;

(10) Реакция на сообщение $Report(w)$ полученное по ребру j

if $j \neq in\_branch$ **then**
  $Find\_count = Find\_count - 1$;
  if $w < best\_wt$ **then**
    $best\_wt = w$;
    $best\_edge = j$;
  report ();
else if $SN = Find$ **then**
  поместить полученное сообщение в конец очереди;
else if $w > best\_wt$ **then**
  change_core ();
else if $w = best\_wt = \infty$ **then**
  halt ();

(11) **procedure** change_core

if $SE[best\_edge] = Branch$ **then**
отправить сообщение $Change\ core$ по ребру $best\_edge$;
else
отправить сообщение $Connect(LN)$ по ребру $best\_edge$;
$SE[best\_edge] = Branch$;

(12) Реакция на сообщение $Change\ core$

change_core ();

3 Параллельный алгоритм поиска минимального остовного дерева в графе

Приведенное описание алгоритма GHS, взятое из статьи [5], является неполным с точки зрения параллельной реализации, оно представляет собой лишь формальное описание и анализ необходимых высокоуровневых действий, которые должны выполняться в каждой вершине.

Для разработки параллельного алгоритма решения задачи MST на основе алгоритма GHS необходимо обоснованно выбрать и разработать набор приемов, разрешить ряд проблем.

Реализация алгоритма производилась на языке C++ с использованием библиотеки MPI [21]. При реализации на суперкомпьютере количество вершин в графе значительно больше, чем MPI-процессов, поэтому в памяти каждого процесса обычно хранится большое количество вершин и вся относящаяся к ним информация. Все вершины графа блоками распределяются последовательно по процессам. Локальная часть графа в каждом процессе хранится в формате CRS (англ. Compressed Row Storage).

3.1 Предварительная обработка исходного графа

Прежде чем искать в графе минимальное остовное дерево, над графом проводится предварительная обработка: из графа удаляются петли и кратные ребра. Удаление кратных ребер используется для выполнения условия алгоритма GHS, которое заключается в том, что все ребра должны быть уникальны. Время, затрачиваемое на предварительную обработку, не учитывается в общем времени работы алгоритма.

3.2 Базовая версия

В начале работы над алгоритмом разработана базовая версия. В каждом MPI-процессе поддерживается очередь, в которую вершины могут откладывать сообщение, если это необходимо. Для ускорения работы реализована агрегация сообщений, для каждого возможного процесса-получателя в каждом процессе заводится отдельный буфер. Схема реализации базовой версии представлена на рис. 3.

```
Input: local_G — локальная часть графа
Output: локальная часть минимального остовного дерева графа

while True do
    /* прием сообщений и запись в очередь */
    read_msgs ();
    /* обработка очереди, отправка сообщений (запись в буфер)
     */
    if time_to_process_queue then
        process_queue ();
    if time_to_send then
        /* отправка всех накопленных сообщений */
        send_all_bufs ();
    check_finish (); /* проверка на завершение при помощи
     MPI_Allreduce */
```

Рис. 3. Схема реализации базовой версии параллельного алгоритма построения MST с использованием библиотеки MPI; выполняется параллельно на каждом MPI-процессе.

В сообщениях кроме необходимой для работы алгоритма информации содержится также служебная информация: номер вершины-отправителя и номер вершины-получателя, а также тип сообщения.

Важно отметить, что алгоритм GHS требует, чтобы исходный граф был связным. В разработанном автором алгоритме это необязательно,

так как алгоритм будет работать до тех пор, пока в сети не наступит состояние «тишины», когда все очереди пусты, все сообщения обработаны, и в сети нет недоставленных сообщений. Таким образом, разработанный алгоритм позволяет находить не только минимальное остовное дерево в связном графе, но также и минимальный остовный лес в графе, с любым количеством компонент связности.

Так как алгоритм GHS требует, чтобы веса всех ребер графа были различными, к обычному весу ребра добавляется специальный идентификатор $special\_id$. Для каждого ребра e графа $special\_id$ вычисляется следующим образом: пусть u и v вершины, инцидентные ребру e, тогда $special\_id$ в битовом представлении равен подряд записанным битовым представлениям $min(u, v)$, $max(u, v)$. При такой организации, даже если во входном графе есть два различных ребра с одинаковым весом, алгоритм будет работать корректно.

3.3 Поиск локальных ребер

Когда MPI-процесс принял входящее сообщение, необходимо найти ребро (индекс ребра в списке локальных ребер), по которому оно пришло, то есть по двум вершинам (отправителю и получателю) в локальном списке ребер найти индекс ребра, которое образуют эти вершины. Поиск необходим для того, что может потребоваться изменение локальных данных, связанных с этим ребром.

В базовой версии для этой операции используется линейный поиск, при котором перебираются все ребра, инцидентные вершине-получателю, если вершина на другом конце ребра равна вершине-отправителю, то нужное ребро найдено.

Первым возможным вариантом оптимизации этой операции была

сортировка у каждой вершины исходного графа всех инцидентных ребер в порядке увеличения номеров вершин на противоположном конце ребра. При таком подходе необходимо вначале работы алгоритма потратить немного времени на сортировку, однако во время работы алгоритма можно использовать бинарный поиск вместо линейного. Такой подход дает небольшой выигрыш по производительности.

Вторая рассмотренная оптимизация — хеширование. Вместо сортировки и бинарного поиска можно в каждом процессе создать хеш-таблицу. Пусть u — вершина-отправитель сообщения, v — вершина-получатель, каждый из номеров — машинное слово длиной 32 бита. Определим хеш-функцию $get\_hash(u, v)$ как

$$((u \ll 32) \mid v) \textbf{ mod } hash\_table\_size,$$

где $\ll$ — битовый сдвиг влево, $\mid$ — побитовое или, **mod** — остаток от деления, $hash\_table\_size$ — размер хеш-таблицы (в несколько раз больше, чем количество локальных ребер). На рис. 4 изображена схема инициализации хеш-таблицы.

На рис. 5 показано, как работает поиск локального ребра при помощи хеширования. Такой метод хеширования называется *линейное исследование и вставка* [22]. Таким образом, найти идентификатор локального ребра по паре смежных вершин можно за $O(1)$, но предварительно нужно создать и заполнить хеш-таблицу. Эта процедура входит в инициализацию алгоритма, занимает крайне незначительное время и не учитывается во времени работы алгоритма.

Input:
$hash\_table$ — пустая хеш-таблица,
$local\_G$ — локальная часть графа
Output: заполненная хеш-таблица $hash\_table$

$hash\_table[i] = \infty; 1 \leqslant i \leqslant hash\_table\_size$
for $u \in local\_V$ **do**
  for $id \in [0, degree(u))$ **do**
    v = номер вершины, находящейся на другом конце ребра,
     стоящего на позиции id среди всех ребер вершины u;
    $cell\_num = get\_hash(u, v)$;
    while $hash\_table[cell\_num] \neq \infty$ **do**
      $cell\_num = cell\_num - 1$;
      if $cell\_num < 0$ **then**
        $cell\_num = cell\_num + hash\_table\_size$;
    $hash\_table[cell\_num] = id$;

Рис. 4. Инициализация хеш-таблицы $hash\_table$ на каждом MPI-процессе.

Input:
$hash\_table$ — хеш-таблица,
$source$ — вершина-отправитель сообщения,
$destination$ — вершина-получатель сообщения
Output: индекс соответствующего локального ребра

$cell\_num = get\_hash(destination, source)$;
$value = hash\_table[cell\_num]$;
while $value = \infty$ *or* $not(edge[value].u =$
 $destination$ *and* $[value].v = source)$ **do**
  $cell\_num = cell\_num - 1$;
  if $cell\_num < 0$ **then**
    $cell\_num = cell\_num + hash\_table\_size$;
  $value = hash\_table[cell\_num]$;
return $value$

Рис. 5. Поиск индекса локального ребра по паре смежных вершин $destination$ и $source$.

3.4 Отдельная обработка сообщений типа *Test*

Сообщения некоторых типов (*Connect*, *Test* и *Report*) не всегда возможно обработать сразу, так как для их обработки необходимо выполнение различных условий. Условие будет выполняться, когда изменятся некоторые данные, а для изменения конкретных данных необходимо дождаться определенного сообщения. Поэтому возникают ситуации, когда сообщение нужно отложить, а затем пытаться обработать снова. Когда оно будет обработано — неизвестно.

```
Input: local_G — локальная часть графа
Output: локальная часть минимального остовного дерева графа

while True do
    /* прием сообщений и запись в очереди */
    read_msgs ();
    /* обработка основной очереди */
    if time_to_process_main_queue then
        process_main_queue ();
    /* обработка очереди с сообщениями типа Test */
    if time_to_process_test_queue then
        process_test_queue ();
    if time_to_send then
        /* отправка всех накопленных сообщений */
        send_all_bufs ();
    check_finish (); /* проверка на завершение при помощи
     MPI_Allreduce */
```

Рис. 6. Схема реализации параллельного алгоритма построения MST с отдельной очередью для сообщений типа *Test*; выполняется параллельно на каждом MPI-процессе.

В процессе исследования работы алгоритма выявлено, что сообщения типа *Test* составляют значительную часть от всех сообщений. Оказалось, что выгодно организовать отдельную очередь для сообщений типа *Test*,

и обрабатывать ее значительно реже, чем основную очередь. Схема реализации алгоритма с отдельной очередью для сообщений типа *Test* представлена на рис. 6.

3.5 Оптимизация длины сообщений

Для достижения максимально возможной производительности программной реализации на распределенной вычислительной системе необходимо минимизировать объем передаваемых по сети данных. Поэтому важно, чтобы структура, которая хранит передаваемое сообщение, занимала как можно меньший объем памяти.

Сначала проведена группировка сообщений на «короткие» (*Connect*, *Accept*, *Reject*, *Change core*) и «длинные» (*Initiate*, *Test*, *Report*). Основное отличие — в «длинных» сообщениях передается вес, а он занимает значительный объем памяти.

В начале каждой структуры, как для «длинных», так и для «коротких» сообщений, хранится битовая маска размером 16 бит (на самом деле, необходимо только 9 бит: 3 бита на тип сообщения, 5 бит на уровень фрагмента, 1 бит на состояние вершины). Далее, в каждой из структур, хранится номер вершины-отправителя и вершины-получателя (номер — машинное слово длиной 32 бита). В длинных сообщениях далее хранится расширенный вес ребра — $special\_id$ и сам вес.

В действительности, задача поиска минимального остовного дерева редко встречается в графах, у которых много ребер с одинаковым весом.

Также реализована следующая оптимизация: вместо того, чтобы хранить в расширенном весе $special\_id$ (склейка двух номеров вершин, суммарно 64 бита), можно хранить минимальный из номеров MPI-процессов, хранящих данное ребро, предварительно проверив, что на каждом процессе

веса всех ребер различны. Действительно, если на каждом процессе веса всех ребер различны, то два различных ребра с одинаковым весом могут находиться только на разных процессах, но тогда, чтобы понять, что такие ребра различны, хватит номеров соответствующих процессов.

3.6 Параметры разработанного алгоритма

На работу программной реализации разработанного алгоритма влияют параметры, модификация которых приводит к изменению производительности:

- BUF_SIZE — размер буфера для агрегации сообщений, которые требуется отправить, в байтах (по умолчанию — 10000 байт),
- $SENDING\_FREQUENCY$ — отвечает за то, как часто сбрасывать буферы с сообщениями (по умолчанию — через каждые 5 итераций цикла $while$),
- $CHECK\_FREQUENCY$ — отвечает за то, как часто обрабатывать очередь с сообщениями типа $Test$ (по умолчанию — через каждые 20 итераций цикла $while$),
- $EMPTY\_ITER\_CNT\_TO\_BREAK$ — отвечает за то, как часто проводить проверку на завершение (по умолчанию — через каждые 100000 итераций цикла $while$),
- $HASH\_TABLE\_SIZE$ — размер хеш-таблицы, измеряется в количестве элементов. По умолчанию равен $local\_actual\_m * 5 * 11/13$, где $local\_actual\_m$ — количество локальных ребер в MPI-процессе после удаления кратных ребер и петель.

4 Исследование разработанного алгоритма на кластере «Ангара-К1» и суперкомпьютере BlueGene/P

Разработка и исследование алгоритма проводились на кластере «Ангара-К1» с сетью «Ангара». Реализация алгоритма также исследовалась на суперкомпьютере IBM Blue Gene/P.

Для оценки производительности алгоритма используются RMAT графы [23]. RMAT графы хорошо моделируют реальные графы из социальных сетей и Интернета, а также являются достаточно сложными для анализа. В данной работе рассматриваются RMAT графы со средней степенью вершины 32, количество вершин является степенью двойки. В таком графе имеется одна большая связная компонента и некоторое количество небольших связных компонент. Все графы генерируются случайным образом (вес ребра — вещественное число, в интервале (0, 1)). Масштаб графа — число, задающее количество вершин в графе. Если n — масштаб графа, то 2^n — количество вершин в этом графе. Граф с масштабом n обозначается в дальнейшем RMAT-n.

Во время запусков реализации разработанного алгоритма используются заданные по умолчанию значения параметров алгоритма, перечисленные в подразделе 3.6.

4.1 Описание вычислительных систем

4.1.1 Кластер «Ангара-К1»

Кластер «Ангара-К1» состоит из 36 вычислительных узлов, объединенных сетью «Ангара» [24], [25]. Топология сети — 3D-тор 3x3x4. Объем

оперативной памяти каждого узла — 64 Гб. Кластер состоит из двух типов узлов. 24 узла имеют в составе по 2 процессора Intel Xeon E5-2630 (6 ядер, 2.3 ГГц). 12 оставшихся узлов — процессор Intel Xeon E5-2660 (8 ядер, 2.2 ГГц).

4.1.2 Суперкомпьютер IBM BlueGene/P

Суперкомпьютер IBM Blue Gene/P [26] — вычислительная система, которая состоит из двух стоек, включающих 8192 процессорных ядер (2048 четырехъядерных вычислительных узлов), с пиковой производительностью 27,9 терафлопс (27,8528 триллионов операций с плавающей точкой в секунду). Вычислительный узел включает в себя четырехъядерный процессор, 2 ГБ оперативной памяти. Микропроцессорное ядро: PowerPC 450, 850 МГц. Топология сети — 3D-тор.

4.2 Исследование вариантов выполнения поиска локального ребра

На рис. 7 показано время работы алгоритма в зависимости от количества узлов и варианта организации поиска локального ребра. Вариант с хешированием оказался самым производительным, поэтому выбран для итоговой версии.

4.3 Сравнение производительности различных версий алгоритма

Для тестирования в данном подразделе использовался граф RMAT масштаба 23 (RMAT-23). Количество MPI-процессов на узел — 8.

На рис. 8 показано, как изменялось время (в секундах) работы

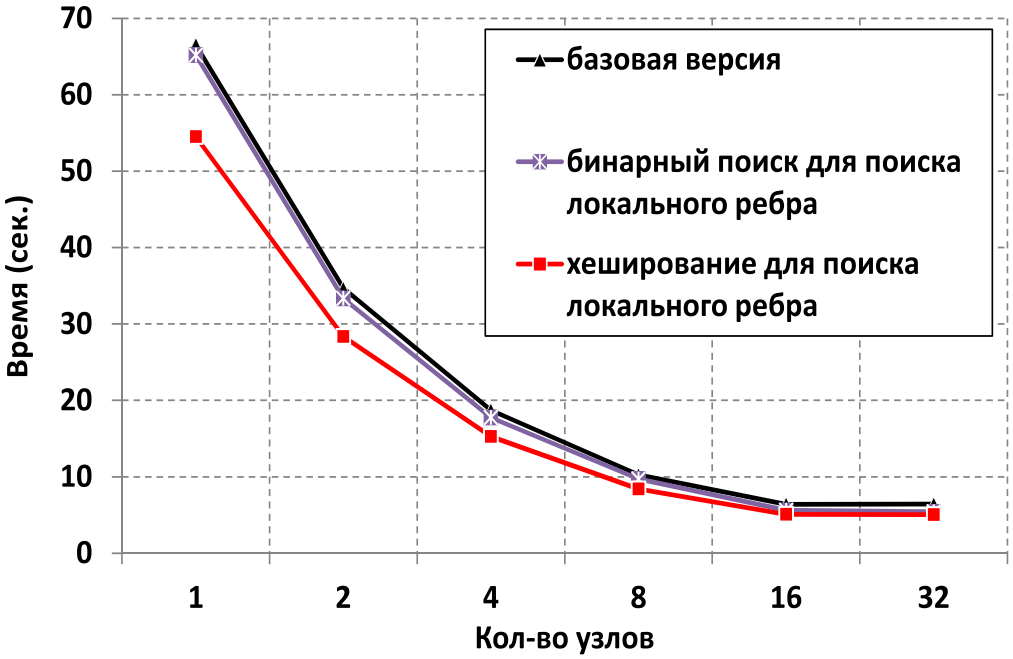

Рис. 7. Время работы. Сравнение версий с разными методами поиска локального ребра. Кластер «Ангара-К1», 8 MPI-процессов на узел, граф RMAT-23.

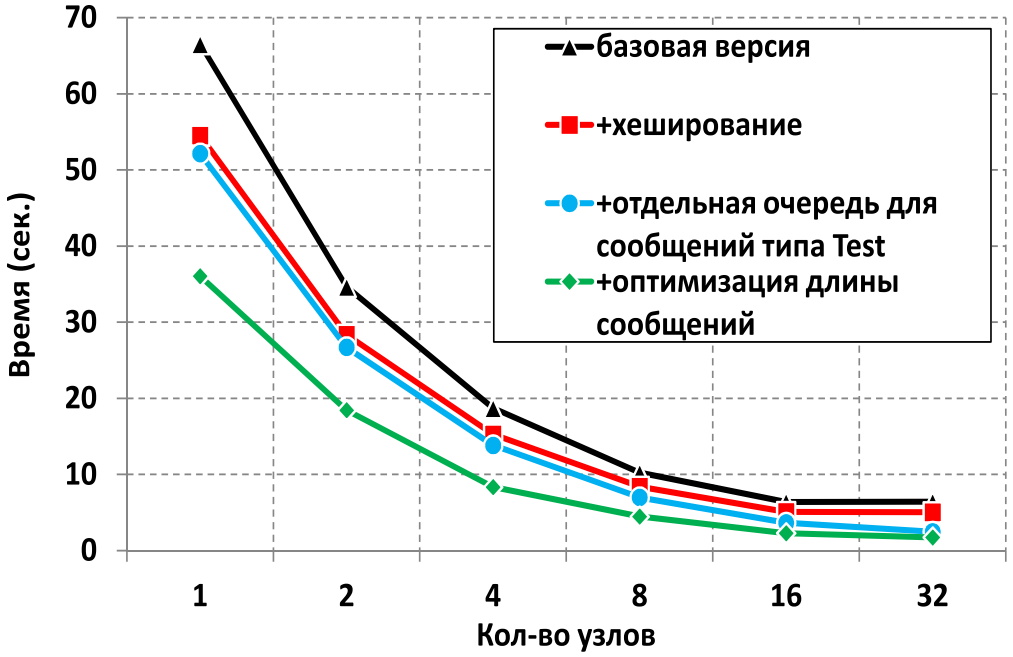

Рис. 8. Время работы. Сравнение версий: от базовой до итоговой (со всеми оптимизациями). Кластер «Ангара-К1», 8 MPI-процессов на узел, граф RMAT-23.

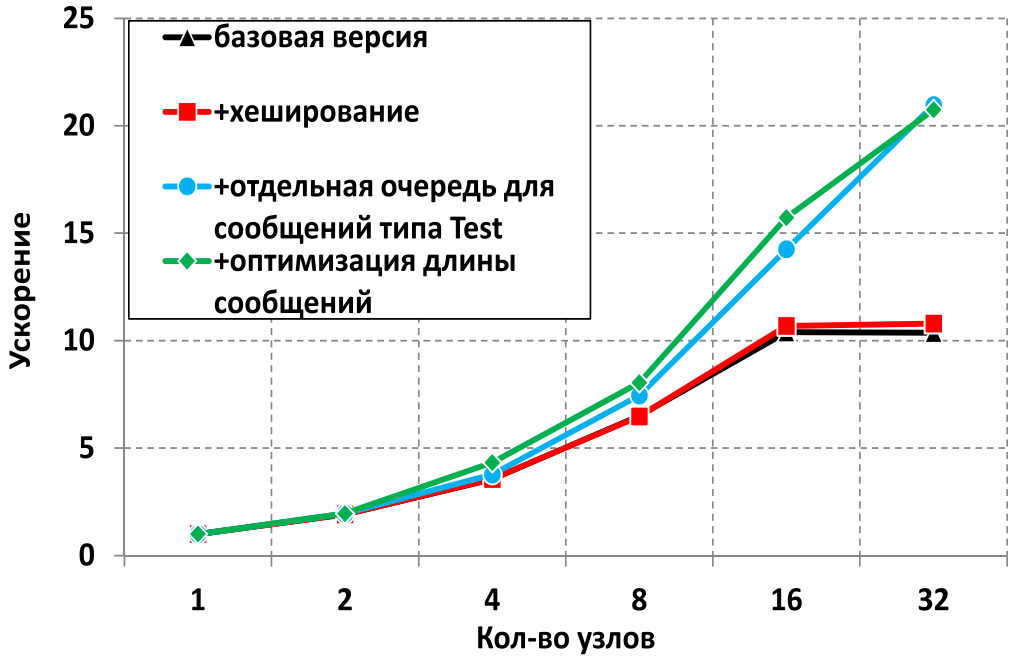

Рис. 9. Масштабируемость. Сравнение версий: от базовой до итоговой. Кластер «Ангара-К1», 8 MPI-процессов на узел, граф RMAT-23.

программы, по мере добавления оптимизаций, описанных в подразделах 3.2, 3.3, 3.4, 3.5. На рис. 9 показана масштабируемость тех же самых запусков, то есть отношение времени решения задачи на одном узле ко времени решения на заданном числе узлов. Рисунки показывают, что на производительность на 32 узлах, а следовательно, и на масштабируемость, сильно повлияла оптимизация, связанная с отдельной обработкой сообщений типа *Test*. Также, на производительность сильно повлияла оптимизация, связанная с уменьшением длины сообщений, которая позволила уменьшить время выполнения итоговой версии алгоритма на любом количестве узлов (от 1 до 32) приблизительно на 25%.

На рис. 10 представлены результаты профилирования варианта программы с одной оптимизацией поиска локального ребра, а на рис. 11 — результаты профилирования итоговой версии программы.

Из приведенных рисунков видно, что при запуске на небольшом

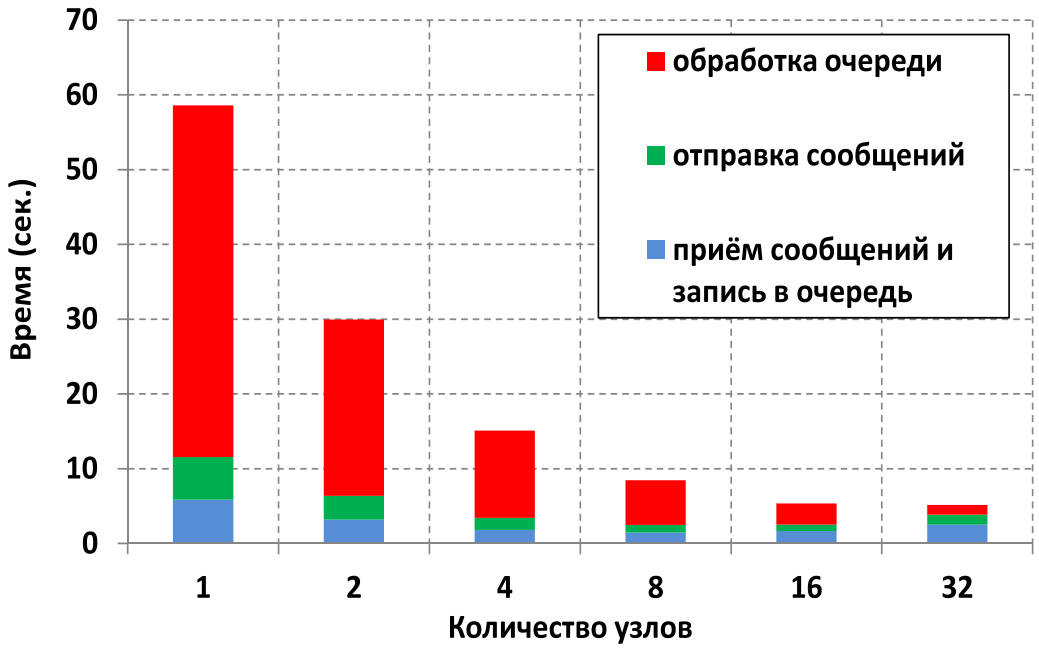

Рис. 10. Результаты профилирования версии с одной оптимизацией — хешированием. Кластер «Ангара-К1», 8 MPI-процессов на узел, граф RMAT-23.

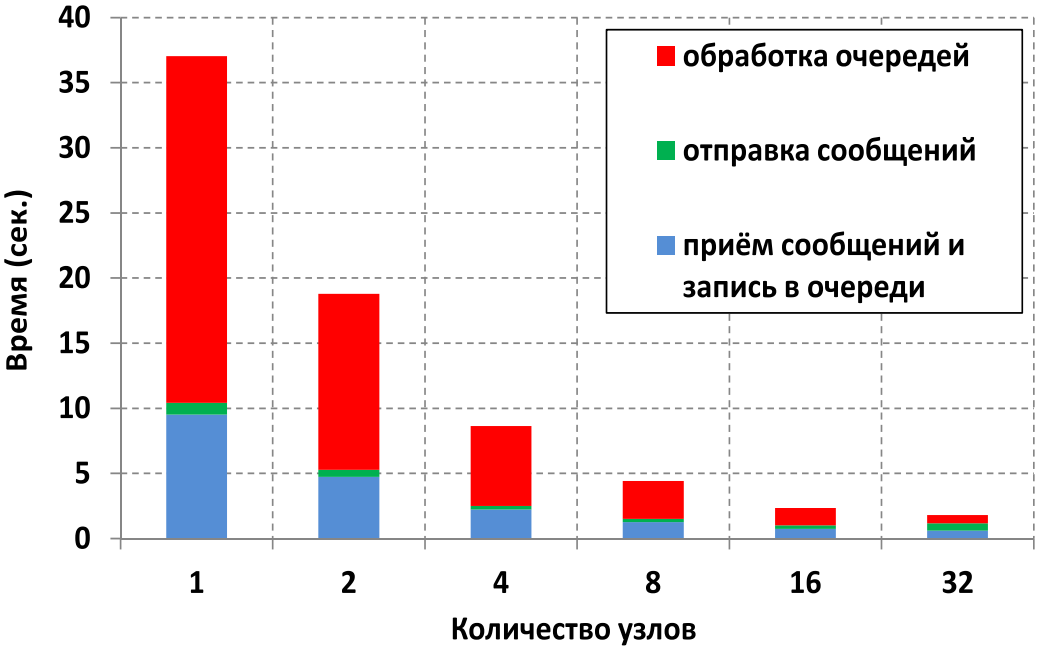

Рис. 11. Результаты профилирования итоговой версии. Кластер «Ангара-К1», 8 MPI-процессов на узел, граф RMAT-23.

количестве узлов, большая часть времени уходит на обработку очередей. Некоторые сообщения обрабатываются повторно, в том числе сообщения типа *Test*, поэтому в итоговой версии алгоритма доля обработки очередей в общем времени работы алгоритма ниже, чем в варианте только с хешированием. Для итоговой версии при запуске на 32 узлах, на обработку очередей, прием и отправку сообщений уходит примерно одинаковое количество времени.

4.4 Исследование производительности алгоритма

На рис. 12 представлена масштабируемость (ускорение) итоговой версии на графе RMAT масштаба 24 (RMAT-24).

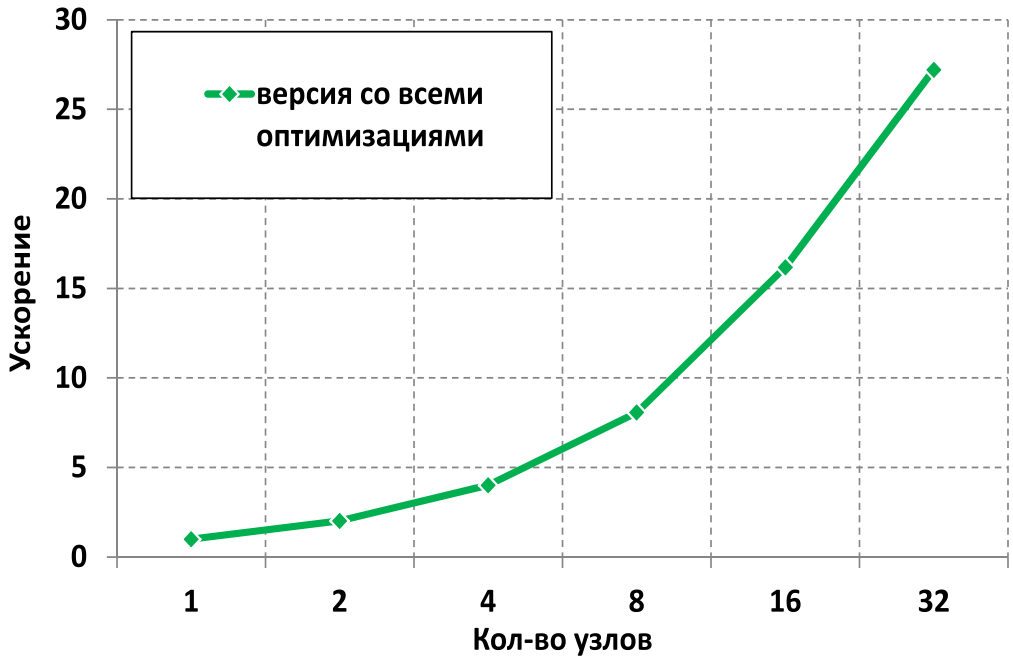

Рис. 12. Масштабируемость. Итоговая версия. Кластер «Ангара-К1», 8 MPI-процессов на узел, граф RMAT-24.

Масштаб 24 взят потому, что RMAT-24 — это наибольший граф, который помещается в память одного узла кластера «Ангара-К1», размер такого графа составляет приблизительно 6.5 ГБ. Остальная оперативная

память вычислительного узла необходима для организации работы реализации. В частности, большой объем памяти необходим для организации хеш-таблицы. Рисунок показывает, что масштабируемость близка к линейной.

Средний размер коммуникационных сообщений в зависимости от периода работы итоговой версии алгоритма показан на рис. 13 . Под размером сообщения здесь понимается агрегированное сообщение (накопленное), которое отправлено в сеть. Значение параметра агрегации BUF_SIZE составляет 20000 байт. Рисунок показывает, что с увеличением количества вычислительных узлов размер сообщений уменьшается. Для 32 узлов сообщения являются короткими, их размер не превосходит 2 КБ. Также видно, что размер сообщений зависит от периода работы алгоритма.

Данный рисунок позволяет высказать утверждение, что разработанный алгоритм предъявляет высокие требования к коммуникационной сети, так как эффективно передавать короткие сообщения возможно в высокоскоростной и высокореактивной сети с низкой задержкой и высоким темпом передачи коротких сообщений.

Время работы на RMAT графах разного размера на 32 узлах кластера «Ангара-К1» представлено на рис. 14 . RMAT-29 — максимальный граф, который поместился в памяти 32 узлов, он занимает суммарно 205 ГБ. Необходимо заметить, что разработанная реализация алгоритма решения задачи MST является масштабируемой по памяти, то есть при увеличении количества вычислительных узлов можно увеличивать размер обрабатываемого графа.

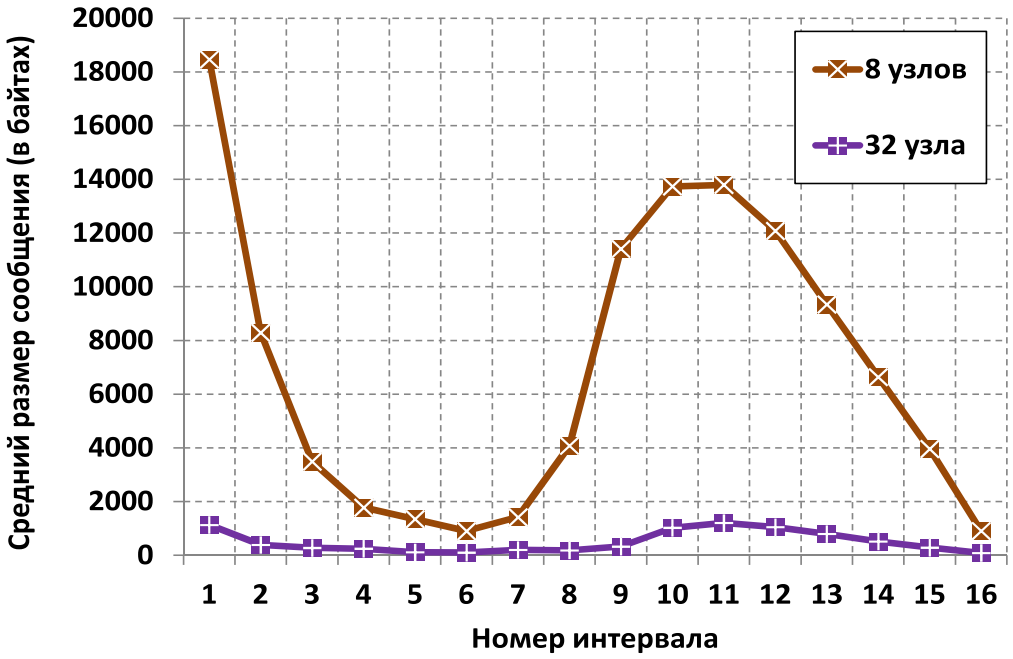

Рис. 13. Средний размер коммуникационных сообщений в байтах в зависимости от номера интервала (общее время работы алгоритма разбито на равные интервалы). Кластер «Ангара-К1», 8 MPI-процессов на узел, граф RMAT-23.

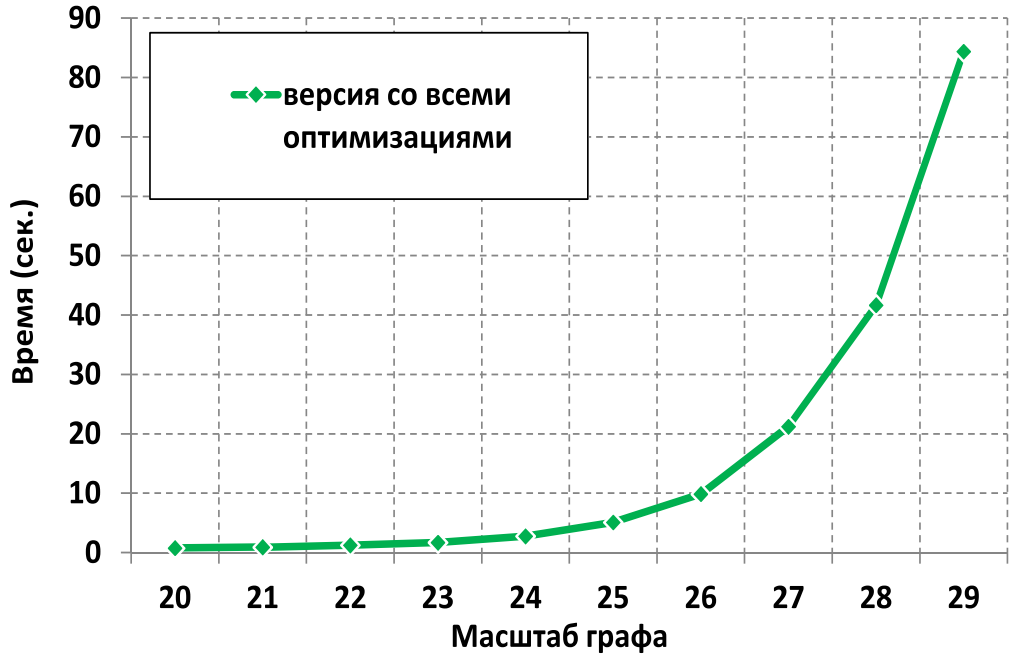

Рис. 14. Время работы итоговой версии на графах разного размера. 32 узла кластера «Ангара-К1». 8 MPI-процессов на узел.

4.4.1 Производительность на суперкомпьютере IBM BlueGene/P

На рис. 15 показано время работы итоговой версии в секундах, в зависимости от количества узлов. Для тестирования использовался граф RMAT масштаба 24. Количество MPI-процессов на узел — 4. На 16 и меньшем количестве узлов граф RMAT-24 не запускается, так как не хватает памяти.

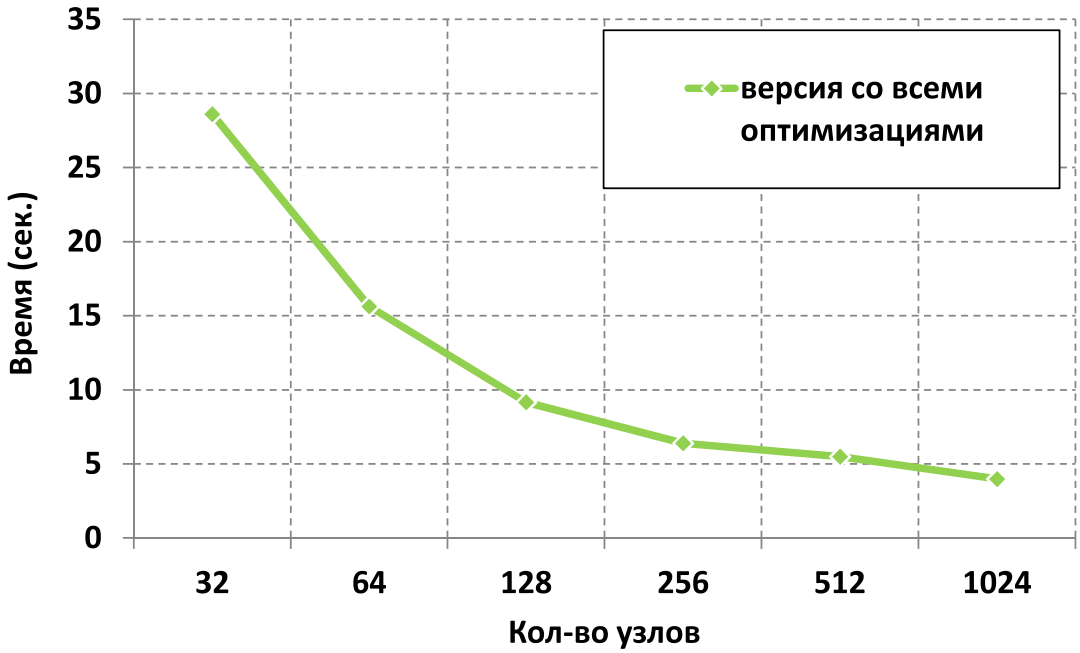

Рис. 15. Время работы. Итоговая версия. Blue Gene/P, 4 MPI-процесса на узел, граф RMAT-24.

На рис. 16 представлена масштабируемость итоговой версии (ускорение в зависимости от количества узлов).

На BlueGene/P наблюдается ускорение при решении задачи на большом числе узлов, однако максимальное ускорение на 1024 узлах по отношению ко времени на 32 узлах составляет 32, а реально полученное — 7.2, что является не очень хорошим результатом. При этом время решения задачи на графе RMAT-24 на кластере «Ангара-К1» составляет 2.89 секунд, а на суперкомпьютере BlueGene/P — 3.98 секунд. Для получения

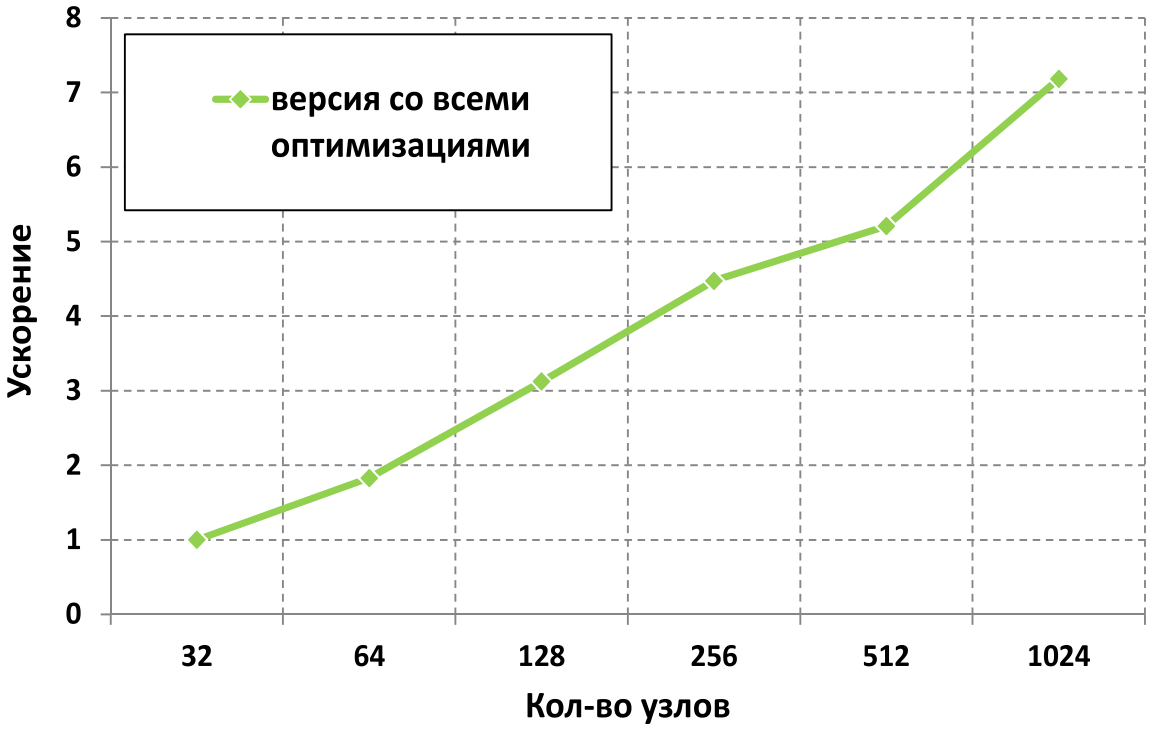

Рис. 16. Масштабируемость. Итоговая версия. Blue Gene/P, 4 MPI-процесса на узел, граф RMAT-24.

лучших результатов на большом числе узлов требуется дополнительная работа.

5 Заключение

В работе представлен разработанный параллельный алгоритм поиска минимального остовного дерева (леса) в графе для вычислительных систем с распределенной памятью, а также программная реализация данного алгоритма. Реализация выполнена с использованием библиотеки передачи сообщений MPI.

Проведено экспериментальное исследование разработанного алгоритма на кластере «Ангара-К1» и суперкомпьютере IBM Blue Gene/P. Показано, что предложенные оптимизации позволили ускорить алгоритм (по сравнению с базовой версией) на 32 узлах кластера «Ангара-К1» в 3.7 раза на RMAT графе с 2^{23} вершин. На RMAT графе с 2^{24} вершин на 32 узлах кластера «Ангара-К1» достигнуто ускорение решения задачи 27.2 по отношению ко времени выполнения на одном узле. Продемонстрирована масштабируемость разработанного алгоритма при реализации на массивно-параллельной системе IBM Blue Gene/P.

Решение задачи для графа RMAT с 2^{29} вершин и 2^{34} ребер (для хранения такого графа необходимо приблизительно 205 ГБ памяти) получено за 84 секунды на 32 узлах кластера «Ангара-К1».

Некоторые результаты работы опубликованы автором в статье [27].

Литература

[1] Ультраконсервативные элементы у простейших из надтипа Alveolata / Рубанов Л.И., Селиверстов А.В., Зверков О.А. [и др.] // Современные информационные технологии и ИТ-образование. 2015. Т. 2. С. 581–585. URL: http://lab6.iitp.ru/ru/pub/ru_sitito_2015_rszl.pdf (дата обращения: 10.05.2016).

[2] Алгоритмы: построение и анализ / Кормен Т., Лейзерсон Ч., Ривест Р. [и др.]; под ред. И. В. Красикова. 2 изд. Вильямс, 2005. 1296 с.

[3] Eisner Jason. State-of-the-Art Algorithms for Minimum Spanning Trees. 1997. URL: https://www.cs.jhu.edu/~jason/papers/eisner.mst-tutorial.pdf (дата обращения: 10.05.2016).

[4] Borůvka Otakar. O jistém problému minimálním // Práce mor. přírodověd. spol. v Brně III. 1926. Т. 3. с. 37–58.

[5] Gallager Robert G., Humblet Pierre A., Spira P. M. A distributed algorithm for minimum-weight spanning trees // ACM Transactions on Programming Languages and Systems. 1983. Т. 5. С. 66–77. URL: http://www.cs.tau.ac.il/~afek/p66-gallager.pdf (дата обращения: 10.05.2016).

[6] Awerbuch Baruch. Optimal Distributed Algorithms for Minimum Weight Spanning Tree, Counting, Leader Election, and Related Problems // Proceedings of the 19th ACM Symposium on Theory of Computing (STOC). New York, 1987.

[7] Prim R. C. Shortest connection networks and some generalizations // Bell System Technical Journal. 1957. Т. 36. С. 1389–1401.

[8] Kruskal Joseph. B. On the Shortest Spanning Subtree of a Graph and the Traveling Salesman Problem // AMS. 1956. T. 7. C. 48–50.

[9] Chazelle Bernard. A Minimum Spanning Tree Algorithm with Inverse-Ackermann Type Complexity // Journal of the ACM. 2000. T. 47. C. 1028–1047.

[10] Bader David A., Cong Guojing. Fast shared-memory algorithms for computing the minimum spanning forest of sparse graphs // Journal of Parallel and Distributed Computing. 2006. T. 66. c. 1366–1378.

[11] Колганов А.С. Параллельная реализация алгоритма поиска минимальных остовных деревьев с использованием центрального и графического процессоров // Параллельные вычислительные технологии (ПаВТ). 2016. URL: http://omega.sp.susu.ru/PaVT2016/short/054.pdf (дата обращения: 11.05.2016).

[12] Hardware and Software Implementations of Prim's Algorithm for Efficient Minimum Spanning Tree Computation / Artur Mariano, Dongwook Lee, Andreas Gerstlauer [и др.]. URL: http://users.ece.utexas.edu/~gerstl/publications/iess13.prims.pdf (дата обращения: 11.05.2016).

[13] Wang Wei. Design and Implementation of GPU-Based Prim's Algorithm // I.J. Modern Education and Computer Science. 2011. URL: http://www.mecs-press.org/ijmecs/ijmecs-v3-n4/IJMECS-V3-N4-8.pdf (дата обращения: 11.05.2016).

[14] An approach to parallelize Kruskal's algorithm using Helper Threads / Anastasios Katsigiannis, Nikos Anastopoulos, Konstantinos Nikas

[и др.]. URL: http://www.cslab.ntua.gr/~knikas/files/papers/mtaap12kruskal.pdf (дата обращения: 11.05.2016).

[15] Piskas Georgios. Parallelizing the Minimum Spanning Tree Problem. 2014. URL: http://www.gpiskas.com/pdf/kruskal_mpi.pdf (дата обращения: 11.05.2016).

[16] Loncar Vladimir, Skrbic Srdjan. Parallel implementation of minimum spanning tree algorithms using MPI. 2012. URL: http://www.uni-obuda.hu/users/szakala/CINTI%202012/6_cinti2012.pdf (дата обращения: 11.05.2016).

[17] Loncar Vladimir, Skrbic Srdjan, Balaz Antun. Parallelization of Minimum Spanning Tree Algorithms Using Distributed Memory Architectures. 2014. URL: http://www.scl.rs/papers/Loncar-TET-Springer.pdf (дата обращения: 11.05.2016).

[18] Sireta Alexis. Comparison of parallel and distributed implementation of the MST algorithm. 2016. URL: http://rp.delaat.net/2015-2016/p41/report.pdf (дата обращения: 11.05.2016).

[19] Ramaswamy Swaroop Indra, Patki Rohit. Distributed Minimum Spanning Trees. 2015. URL: http://stanford.edu/~rezab/classes/cme323/S15/projects/distributed_minimum_spanning_trees_report.pdf (дата обращения: 11.05.2016).

[20] Robert Ryan McCune Tim Weninger Gregory Madey. Thinking Like a Vertex: a Survey of Vertex-Centric Frameworks for Distributed Graph Processing // ACM Comput. Surv. 2015. T. 48.

[21] Message Passing Interface Homepage. URL: http://www.mpi-forum.org (дата обращения: 10.05.2016).

[22] Кнут Дональд Эрвин. Искусство программирования. 2 изд. Вильямс, 2007. Т. 3. С. 563–565.

[23] Chakrabarti Deepayan, Zhan Yiping, Faloutsos Christos. R-MAT: A Recursive Model for Graph Mining. URL: http://www.cs.cmu.edu/~christos/PUBLICATIONS/siam04.pdf (дата обращения: 10.05.2016).

[24] Первое поколение высокоскоростной коммуникационной сети «Ангара» / Симонов А.С., Макагон Д.В., Жабин И.А. [и др.] // Наукоемкие технологии. 2014. Т. 15. С. 21–28.

[25] Предварительные результаты оценочного тестирования отечественной высокоскоростной коммуникационной сети Ангара / А.А. Агарков, Т.Ф. Исмагилов, Д.В. Макагон [и др.] // Параллельные вычислительные технологии (ПаВТ). 2016. URL: http://omega.sp.susu.ru/PaVT2016/full/056.pdf (дата обращения: 11.05.2016).

[26] Описание Blue Gene/P. URL: http://hpc.cmc.msu.ru/bgp (дата обращения: 10.05.2016).

[27] А. В. Мазеев А. С. Семенов А. С. Фролов. Сравнение технологий параллельного программирования MPI и Charm++ на примере задачи построения минимального остовного дерева в графе // Comp. nanotechnol. 2015. С. 18–25. URL: http://www.mathnet.ru/links/ccc975549969e25b062433b9099b3dfb/cn48.pdf (дата обращения: 10.05.2016).